**Cognizant DN4.0 deep skilling program**

**Week 1 mandatory hands on exercise**

**Name: Brishti Das**

**Superset id - 6363729**

**Design Pattern and Principle**

**1.**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**Answer Code :**

**Singleton.cs**

using System;

public class Logger

{

private static Logger instance = null;

private Logger()

{

Console.WriteLine("Logger Initialized.");

}

public static Logger GetInstance()

{

if (instance == null)

{

instance = new Logger();

}

return instance;

}

public void Log(string message)

{

Console.WriteLine($"Log: {message}");

}

}

**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Testing Singleton Pattern:\n");

Logger logger1 = Logger.GetInstance();

logger1.Log("First log message.");

Logger logger2 = Logger.GetInstance();

logger2.Log("Second log message.");

if (logger1 == logger2)

{

Console.WriteLine("\nBoth logger1 and logger2 refer to the same instance.");

}

else

{

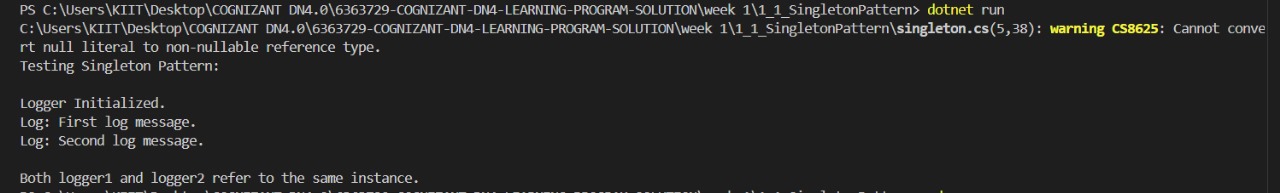
Console.WriteLine("\nDifferent Logger instances detected! Singleton failed.");

}

}

}

**Output** :



**2.**

**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

Answer Code :

**Document.cs**

// IDocument.cs

public interface IDocument

{

void Open();

}

// WordDocument.cs

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a Word document.");

}

}

// PdfDocument.cs

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a PDF document.");

}

}

// ExcelDocument.cs

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening an Excel document.");

}

}

**Factories.cs**

// DocumentFactory.cs

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

public void DisplayFactoryType()

{

Console.WriteLine($"Factory: {this.GetType().Name}");

}

}

// WordDocumentFactory.cs

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

// PdfDocumentFactory.cs

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

// ExcelDocumentFactory.cs

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

**Program.cs**

class Program

{

static void Main()

{

Console.WriteLine("Factory Method Pattern : ");

DocumentFactory wordFactory = new WordDocumentFactory();

Console.WriteLine($"Factory: {wordFactory.GetType().Name}");

IDocument wordDoc = wordFactory.CreateDocument();

Console.WriteLine($"Created: {wordDoc.GetType().Name}");

wordDoc.Open();

Console.WriteLine();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Console.WriteLine($"Factory: {pdfFactory.GetType().Name}");

IDocument pdfDoc = pdfFactory.CreateDocument();

Console.WriteLine($"Created: {pdfDoc.GetType().Name}");

pdfDoc.Open();

Console.WriteLine();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Console.WriteLine($"Factory: {excelFactory.GetType().Name}");

IDocument excelDoc = excelFactory.CreateDocument();

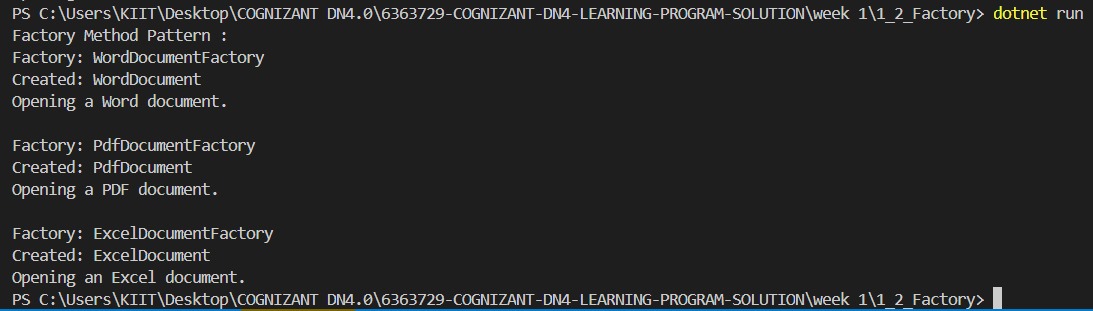
Console.WriteLine($"Created: {excelDoc.GetType().Name}");

excelDoc.Open();

}

}

**Output** :



**Data Structures and Algorithm:**

**3.**

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Answer Code :**

### Big O Notation

Big O notation describes the upper bound of an algorithm's time or space complexity in terms of the input size. It helps us analyze how an algorithm's performance scales as the input grows, allowing us to compare different algorithms and choose the most efficient one for a given problem.

### Search Operation Scenarios

****Best-case****: The ideal scenario where the algorithm finds the element immediately (first element checked for linear search, middle element for binary search).

****Average-case****: The expected performance over all possible inputs.

****Worst-case****: The scenario where the algorithm takes the maximum time (element not present or at the end for linear search)

**Product.cs**

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

public Product(int id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

}

**Search**.**cs:**

using System;

public class SearchAlgorithms

{

public static Product LinearSearch(Product[] products, string name)

{

foreach (var product in products)

{

if (product.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))

return product;

}

return null;

}

public static Product BinarySearch(Product[] products, string name)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int comparison = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

**Program.cs**

// Program.cs

using System;

class Program

{

static void Main()

{

Product[] products = new Product[]

{

new Product(101, "Laptop", "Electronics"),

new Product(245, "Shoes", "Footwear"),

new Product(321, "Book", "Stationery"),

new Product(490, "Headphones", "Electronics"),

new Product(576, "Mobile", "Electronics")

};

Array.Sort(products, (a, b) => a.ProductName.CompareTo(b.ProductName));

string searchItem = "Mobile";

Console.WriteLine("Using Linear Search:");

var result1 = SearchAlgorithms.LinearSearch(products, searchItem);

Console.WriteLine(result1 != null ? $"Found: {result1.ProductName}" : "Product not found");

Console.WriteLine("\n Using Binary Search:");

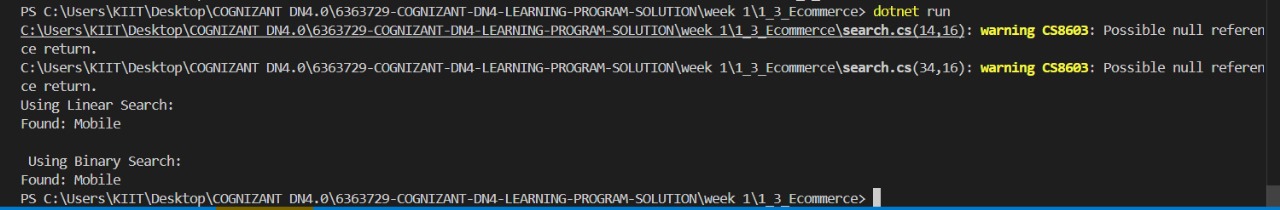
var result2 = SearchAlgorithms.BinarySearch(products, searchItem);

Console.WriteLine(result2 != null ? $"Found: {result2.ProductName}" : "Product not found");

}

}

**Output:**



### 4. Analysis

#### ****Time Complexity****

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| ****Linear Search**** | O(1) | O(n) | O(n) |
| ****Binary Search**** | O(1) | O(log n) | O(log n) |

#### ****2. Best for E-Commerce:-****

****Binary Search**** :

* Catalog is **large & sorted**
* **Frequent searches**
* **Sorting overhead is acceptable**

****Linear Search**** :

* Catalog is **small**
* **Frequent updates** (hard to keep sorted)
* **Memory constraints**

#### ****Production Recommendations****

****Conclusion:**** Binary search is better for large, static datasets, while linear works for small or dynamic data.

**4.**

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Answer Code :**

## 1. Understanding Recursive Algorithms

**Recursion** is a technique where a method calls itself to solve smaller instances of the same problem. It is useful for problems that can be broken down into similar subproblems.

### Benefits of Recursion:

* Reduces code complexity for problems like factorial, Fibonacci, tree traversal, etc.
* Makes the code cleaner and more logical when dealing with repetitive patterns.

**Program.cs**

using System;

using System.Collections.Generic;

namespace ForecastApp

{

class ForecastCalculator

{

private static Dictionary<int, double> cache = new Dictionary<int, double>();

public static double GetForecast(double amount, double growthRate, int period)

{

if (period == 0)

return amount;

if (cache.ContainsKey(period))

return cache[period];

double updatedAmount = amount \* (1 + growthRate);

double forecast = GetForecast(updatedAmount, growthRate, period - 1);

cache[period] = forecast;

return forecast;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Financial Forecast Calculator : ");

Console.Write("Initial investment amount: ");

double initialAmount = double.Parse(Console.ReadLine());

Console.Write("Annual increase rate (like 0.07 for 7%): ");

double growth = double.Parse(Console.ReadLine());

Console.Write("Number of years to forecast: ");

int duration = int.Parse(Console.ReadLine());

double predictedAmount = ForecastCalculator.GetForecast(initialAmount, growth, duration);

Console.WriteLine($"\nPredicted Value After {duration} Years: {predictedAmount:F2}");

Console.WriteLine("\nAnalysis:");

Console.WriteLine("Recursive time complexity without caching: O(n)");

Console.WriteLine("Optimized with caching (memoization): nearly O(1) for repeated periods");

Console.WriteLine("\nPress Enter to exit...");

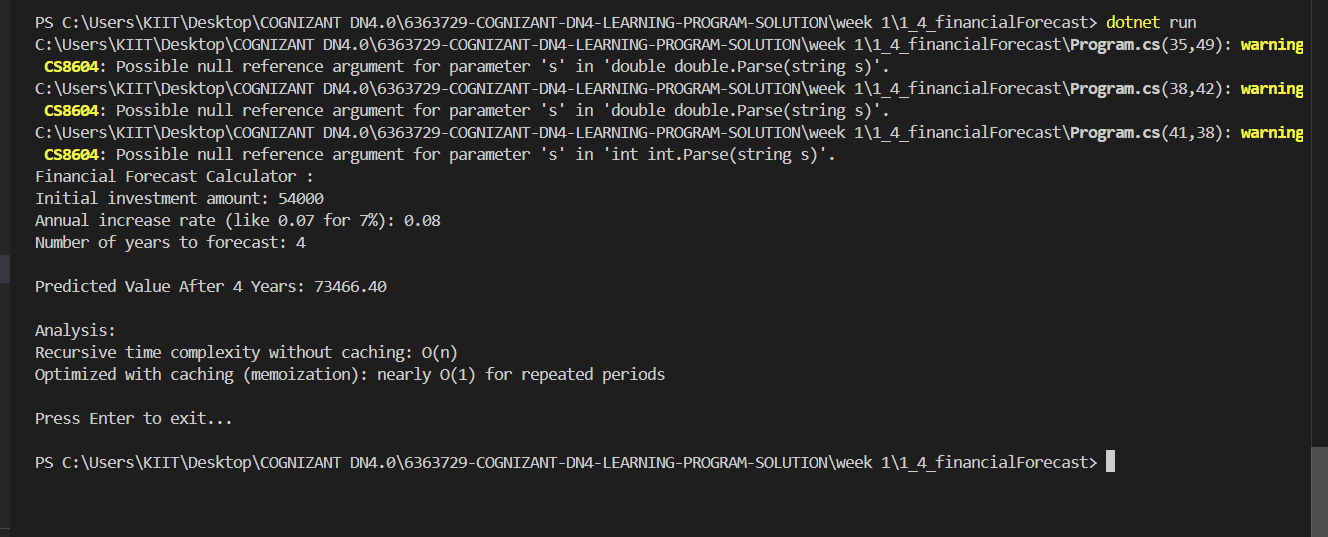
Console.ReadLine();

}

}

}

**Output:**



## 4. Analysis

### Time Complexity

### ****Basic Recursive Version****: O(n) where n is the number of periods.

### Each recursive call reduces the problem size by 1

* Function gets called exactly n times

****Memoized Version****: O(n) time, but with O(n) space complexity

* Each unique (value, rate, period) combination is computed only once
* Subsequent calls return cached results

### Optimization Techniques

* ****Memoization (Caching)****

Stores previously computed results to avoid redundant calculations

Particularly useful if the same parameters are used repeatedly

* ****Iterative Approach****

Can be more efficient than recursion for this simple case: